Model-based mutant equivalence detection using automata language equivalence and simulations

Devroey, Xavier; Perrouin, Gilles; Papadakis, Mike; Legay, Axel; Schobbens, Pierre-Yves; Heymans, Patrick

Published in:
Journal of Systems and Software

DOI:
10.1016/j.jss.2018.03.010

Publication date:
2018

Model-based mutant equivalence detection using automata language equivalence and simulations

Xavier Devroey[a,*], Gilles Perrouin[b,1], Mike Papadakis[c], Axel Legay[d], Pierre-Yves Schobbens[b], Patrick Heymans[b]

[a] *SERG, Delft University of Technology,Postbus 5, 2600 AA Delft, The Netherlands*
[b] *PReCISE, Namur Digital Institute, Faculty of Computer Science, University of Namur, rue Grandgagnage 21, B-5000 Namur,Belgium*
[c] *Interdisciplinary Centre for Security, Reliability and Trust (SnT), SERVAL Team, University of Luxembourg, 29, Avenue J.F Kennedy, L-1855 Luxembourg*
[d] *INRIA Rennes, Campus universitaire de Beaulieu, 35042 Rennes Cedex, France*

Abstract

Mutation analysis is a popular technique for assessing the strength of test suites. It relies on the mutation score, which indicates their fault-revealing potential. Yet, there are mutants whose behaviour is equivalent to the original system, wasting analysis resources and preventing the satisfaction of a 100% mutation score. For finite behavioural models, the Equivalent Mutant Problem (EMP) can be transformed to the language equivalence problem of nondeterministic finite automata for which many solutions exist. However, these solutions are quite expensive, making computation unbearable when used for tackling the EMP. In this paper, we report on our assessment of a state-of-the-art exact language equivalence tool and two heuristics we proposed. We used 12 models, composed of (up to) 15,000 states, and 4,710 mutants. We introduce a random and a mutation-biased simulation heuristics, used as baselines for comparison. Our results show that the exact approach is often more than ten times faster in the weak mutation scenario. For strong mutation, our biased simulations can be up to 1,000 times faster for models larger than 300 states, while limiting the error of misclassifying non-equivalent mutants as equivalent to 8% on average. We therefore conclude that the approaches can be combined for improved efficiency.

Keywords: model-based mutation analysis, automata language equivalence,

[☆] This article is an extension of our previous work [1], published as a short paper in the proceedings of the 10th IEEE International Conference on Software Testing, Verification and Validation (ICST 2017)

[*] Corresponding author
Email addresses: x.d.m.devroey@tudelft.nl (Xavier Devroey), gilles.perrouin@unamur.be (Gilles Perrouin), michail.papadakis@uni.lu (Mike Papadakis), axel.legay@inria.fr (Axel Legay), pierre-yves.schobbens@unamur.be (Pierre-Yves Schobbens), patrick.heymans@unamur.be (Patrick Heymans)
[1] FNRS research associate

random simulations

1. Introduction

Mutation analysis is a technique that injects artificial defects, called *mutations*, into the code under test, yielding *mutants*. Mutants are typically used to evaluate the effectiveness of test suites [2, 3, 4] and to support test generation [5, 6, 3]. The technique is quite popular in research due to the ability of mutants to simulate the behaviour of real faults [2, 7]. There is also evidence showing that tests designed to detect mutants reveal significantly more faults than other test criteria [8, 3, 9].

These characteristics of mutation inspired researchers to apply the method on artefacts other than code and particularly models [3, 10, 4]. The usual advantages of model-based testing technique is the ability to identify defects related to missing functionality or misinterpreted specifications [11] where code-based testing fails [12, 13]. The method has been shown to be practical and can complement existing approaches. For instance, Aichernig *et al.* [14] report that model mutants lead to tests that are able to reveal implementation faults that were found neither by manual tests, nor by the actual operation, of an industrial system.

Despite its potential, mutation analysis faces a number of challenges that currently prevent wider adoption [15, 4]. One of them is the *Equivalent Mutants Problem* (EMP). This problem concerns the identification of the mutants whose behaviour is identical to the original artefact (code or model). Such mutants cannot be distinguished by any test, a situation that raises two issues: (i) they hamper the use of the criterion as a stopping rule by skewing the mutation score measurement (the number of detected mutants divided by the total number of mutants), and (ii) they do not bring any new value to the test generation techniques as they attempt to kill mutants that have no chance to be killed.

In this paper, we focus on the model-based formulation of the EMP, which can be expressed in terms of language equivalence. Language equivalence has been studied by the formal verification community who determined its PSPACE complexity [16] and derived exact equivalence checking algorithms [17, 18]. While potentially helpful, such tools have, to our knowledge, never been used to tackle the EMP. This is the main contribution and novelty of this paper.

In summary, the contributions of this paper are:

- The design of two simulation algorithms relying on random simulations (RS) and biased simulations (BS) that aim at covering infected states [19] (*i.e.*, exploiting syntactical differences between original and mutant models) to improve the chances to distinguish non-equivalent mutants;
- A configurable implementation of our simulations (available at `https://projects.info.unamur.be/vibes/`) that benefits from the fact that simulation can be easily distributed among processor cores;

- The definition of an experimental setup to apply an automata language equivalence tool (ALE) [17] to the EMP. We employed twelve models of varying origins and sizes, from 9 to 15,000 states. We produced 4,710 mutants using seven operators, and considered four mutation orders (one, two, five, ten), according to strong and weak mutation scenarios.

- The assessment of the ALE tool with respect to our baseline algorithms. We measured the speed and accuracy of equivalence detection. The ALE tool is particularly efficient for weak mutation by being, on average, ten times faster than simulations. However, biased simulations perform well for strong mutation on models larger than 300 states: they can be 1,000 times faster. The ratio of tagging non-equivalent mutants as equivalent is 8% for biased simulations and 15% for random ones. To ease reproducibility, all our models and experimental results are available at: `https://projects.info.unamur.be/vibes/mutants-equiv.html`.

This paper extends our previous work [1] on the major following points: the empirical analysis is now performed on 12 models of size up to 15,000 states and 4,710 mutants (instead of 3 models and 1,170 mutants); it adds a new research question to analyse the impact of strong and weak mutation on automata language equivalence performance; finally, we provide statistical significance evidence.

The remainder of the paper is organised as follows. Section 2 presents background information on the models used and language equivalence, while Section 3 details the design of our simulation heuristics and the ALE approach we used. Section 4 describes our empirical assessment and provides some lessons learned. Section 5 covers relevant literature. Finally Section 6, wraps up the paper.

2. Background

In this section we introduce the main formalism, namely, finite transition systems, and the relevance of language equivalence for equivalent mutant detection, that we use throught the paper.

2.1. Transition Systems & Finite Automata

We consider transition systems as a powerful abstract formalism to model system behaviour. We addapt and follow the definition of Baier and Katoen's [20], where atomic propositions have been omitted (we do not consider state internals). Thus, we consider:

Definition 1 (Transition System (TS)). *A TS is a tuple* $(S, Act, trans, i)$ *where* S *is a set of states,* Act *is a set of actions,* $trans \subseteq S \times Act \times S$ *is a non-deterministic transition relation (with* $(s_1, \alpha, s_2) \in trans$*, denoted* $s_1 \xrightarrow{\alpha} s_2$*), and* $i \in S$ *is the initial state.*

To deal with test generation activities, where finite behaviours are sought, we first require the sets S and Act to be finite. To mimic weak and strong mutation scenarios (see Section 3.1), we impose the requirement of stoping the test execution at specific states. These requirements make the non-deterministic finite automata (NFA) sematics be equivalent to our executions. This key observation enables the comparison of our simulations with the ALE tools. In the remainder of this paper, unless otherwise stated, we refer to TSs with such restrictions so that the term can be used interchangeably with NFAs[2].

Definition 2 (Trace). *Let $ts = (S, Act, trans, i)$ be a TS, let $t = (\alpha_1, \ldots, \alpha_n)$ where $\alpha_1, \ldots, \alpha_n \in Act$ be a finite sequence of actions. The trace t is valid iff:*

$$ts \stackrel{(\alpha_1,\ldots,\alpha_n)}{\Longrightarrow}$$

where $ts \stackrel{(\alpha_1,\ldots,\alpha_n)}{\Longrightarrow}$ is equivalent to $\exists s \in S : i \stackrel{(\alpha_1,\ldots,\alpha_n)}{\Longrightarrow} s$, meaning that there exists a non-empty sequence of transitions labelled $(\alpha_1, \ldots, \alpha_n)$ from i to a state s of the TS.

2.2. Equivalent Mutant Problem

In this paper, we focus on the model-based instance of the Equivalent Mutant Problem (EMP). The equivalent mutant problem is a well-known issue in mutation analysis [4, 19, 15]. It stems from the fact that two program variants may exhibit the same behaviour and therefore cannot be distinguished by test cases. This is particularly problematic with respect to both generation and assessment of test suites, since in the former case resources are spent on trying to kill non-killable mutants and in the later case skewing the assessment score (100% of killed mutants is impossible to reach in case of equivalence). Mutant equivalence can take two forms [15]: (a) equivalence between mutants and the original system; (b) equivalence between two mutants (not with the original system). Mutants of case (a) are called *equivalent* while mutants of case (b) are called *duplicate*. In the context of this paper, we focus on mutants that are behaviourally *equivalent* to the original system, *i.e.*, mutants of case (a).

2.3. Automata Language Equivalence & EMP

In our context, the EMP corresponds to a classic problem in automata theory: *Automata Language Equivalence* (ALE). The accepted language of an automaton is formed by all the sequences of actions (words) that can be accepted *i.e.*, starting in the initial state and ending in a final state. Therefore, if a mutant m accepts the same language as the original o (language-equivalent), then there is no trace t that can distinguish the mutant from the original:

$$\forall t, t \in \mathcal{L}(o) \Leftrightarrow t \in \mathcal{L}(m)$$

[2]Our MBT framework, VIBeS, uses TSs as its underlying formalism so we stick to the term "TS" for consistency.

There are various forms of relations that can be defined between two automata in order to determine whether they are language-equivalent. Among them, we can cite bisimulations or trace equivalence [20]. In the recent years, the verification community developed dedicated algorithms, such as bisimulations up to congruence [17] or antichains [18], to address language equivalence. In model-based mutation testing, Aichernig *et al.* investigated language inclusion (but not equivalence) using refinement checking [21] in order to generate mutant-killing test cases.

Although the language equivalence and inclusion problems can be tackled with many techniques, these may face exponential blow-up since they are PSPACE complete [16]. Thus, ending up with a discouraging worst-case complexity. To this end, various heuristics have been proposed, aiming at reducing the average complexity faced in practice. Here, we try to determine the applicability of an exact language equivalence algorithm, in particular the one proposed by Bonchi and Pous [17], at addressing the EMP. This algorithm has been selected due to its availability, reported performance and its ability to handle non-determinism that mutations may incur. In the next section, we also present two baseline algorithms that run generated traces to distinguish original and mutants' behaviours.

3. Mutant Equivalence Analysis

3.1. Strong and Weak Mutation

Elizabeth Jöbstl [22] discussed the conditions, identified by DeMillo and Offutt [23], that must be fulfilled to kill a mutant: (i) "the *necessity condition* says that the state of the mutated program after some execution of the mutated statement must be incorrect with respect to the original program. This implies that the mutated statement must be reached. This is necessary, but not sufficient"; (ii) "the *sufficiency condition* says that the final state of the mutant must differ from the final state of the original program, i.e., the necessary incorrect intermediate state must propagate to an incorrect final state." Satisfying the necessity condition alone is referred to as *weak mutation* [24], while satisfying both is *strong mutation*.

At the model level, our simulations detect an incorrect state if a trace that is valid with respect to the original TS is invalid on the mutant TS, and vice-versa. Indeed, when executed, a trace induces one or more *runs* (alternating sequences of states and actions), depending on the presence of non-determinism. If such a run contains only a prefix of the trace in its sequence (*i.e*, the run is *incomplete*), it is because of the presence of an incorrect state preventing the remaining actions to be fired. If all runs are complete, the original and the mutant are assumed equivalent for this trace. Necessity and sufficiency conditions affect the final states of these runs. For weak mutation, these states can map to any state of the TS. For strong mutation, we need to account for the fact that TSs have no final states. A very frequent example is the modelling of user sessions in which, after a legitimate sequence of actions, the system returns to its initial

state to welcome a new user. This occurs in two thirds of the systems we analyse in Section 4.1.1. This is why we model strong mutation by generating traces whose runs start and end in the same initial state, assimilated to a final state.

The ALE approach uses automata that have explicit initial and final states. For weak mutation, we generate automata in which all states are final, and for strong mutation the initial state is the only final state.

3.2. Automata Language Equivalence (ALE)

For comparison, we selected the ALE approach developed by Bonchi and Pous [17]. It is an extension of the Hopcroft-Karp algorithm to non-deterministic TSs: they introduce a new bisimulation relation, called *up to congruence*, that requires to explore less states compared to the original algorithm; and performs determinisation on-the-fly to avoid building the complete deterministic finite TS. This approach is particularly relevant for the EMP: (i) non-determinism may be introduced locally by mutations (our original models are deterministic), thereby limiting determinisation scope; and (ii) between 0% and 15.5% of our mutants are non-deterministic (see Section 4.1.1).

3.3. Random and Biased Simulation

Our randomized approach to equivalence analysis is straightforward: we generate random traces from the original model and run them on the mutant model and reciprocally. If a trace fails to execute on one of the models, it serves as a counterexample and disproves equivalence. If all runs succeed, then the mutant is considered *probably equivalent* and testers have to decide if they want to perform more simulations or switch to an exact method. Algorithm 1 presents our generic simulation approach: N traces are selected (resp.) from the original model (line 1) and the mutant model (line 7), and executed (resp.) on the mutant model (line 3) and the original model (line 9). In case of non deterministic behaviour, all the possible paths (*i.e.*, runs) are considered for the execution of the trace. If one execution fails, the algorithm stops and returns a *positive* trace such as $(o \stackrel{t}{\Longrightarrow}) \wedge \neg(m \stackrel{t}{\Longrightarrow})$ (line 4) or a *negative* trace such as $\neg(o \stackrel{t}{\Longrightarrow}) \wedge (m \stackrel{t}{\Longrightarrow})$ (line 10) .

This generic simulation algorithm is instantiated through two strategies for trace generation (lines 1 and 7): *Random Simulation* (RS) and *Biased Simulation* (BS). The parameter N is computed using the Chernoff-Hoeffding bound as explained hereafter.

3.3.1. Random Simulation (RS)

Random simulation (RS) assumes a uniform distribution of traces over the model, that is, such traces are selected randomly (*select* call on lines 1 and 7 in Algorithm 1) by accumulating the actions α_i triggered by a random walk of a given length $\leq k$ in the TS. For weak mutation (WM RS), the only constraint is to start the random walk from the initial state i. Strong mutation (SM RS) requires a random walk starting from and ending in i: after few tries, this method (*i.e.*, using a random walk until the initial state i is reached) showed

Algorithm 1 Generic simulation

Require: $o : TS$ {the original system}
$m : TS$ {the mutant to compare to o}
N {total number of traces to generate}
k {trace length}

Ensure: returns a positive or negative trace differentiating m from o or a special value ($none$) if m is equivalent to o.

1: $traceset \leftarrow select(o, \frac{N}{2}, k)$
2: **for all** $t \in tracetset$ **do**
3: **if** $\neg(m \overset{t}{\Longrightarrow})$ **then**
4: **return** $pos(t)$ {if the mutant TS fails to execute t, returns a positive trace t}
5: **end if**
6: **end for**
7: $traceset \leftarrow select(m, \frac{N}{2}, k)$
8: **for all** $t \in traceset$ **do**
9: **if** $\neg(o \overset{t}{\Longrightarrow})$ **then**
10: **return** $neg(t)$ {if the original TS fails to execute t, returns a negative trace t}
11: **end if**
12: **end for**
13: **return** $none$

very poor results on our largest models (we set a timeout of one hour for one equivalence detection) and is therefore not further discussed in this paper.

3.3.2. Biased Simulation (BS)

The biased simulation (BS) approach exploits the basic characteristics of mutation testing: mutations are localised and they create (most of the time) behavioural differences. It assumes that those differences are detected by a trace t which, when executed on the original TS o or on its mutant m, goes through one of the states affected by the mutation. For instance, the transition missing (TMI in Table 2) operator produces a mutant by removing a transition $a \xrightarrow{\alpha_i} b$ from the original TS. The BS approach generates traces in o and m, such that their executions $m \overset{t}{\Longrightarrow}$ or $o \overset{t}{\Longrightarrow}$ cover a or b. Such states, called *infected states*, have been shown to help identifying equivalent mutants at the code level [25, 26] and to speed up mutation analysis at the model level [27]. This motivates us to adopt this strategy in our biased simulation.

In practice, the set of infected states S_{infect} is computed by checking syntactic differences between the original and mutant TSs. It will include: (i) connected states (i.e, states accessible from the initial state) from one model which are not present in the other, and (ii) states with differences in their input/output transitions: in number of transitions or in action names, considering any pair of

states $< s_o, s_m >$ where s_o is a state in the original TS, s_m a state in the mutant TS, such that their names are identical. An alternative is to instrument the mutant generator to keep track of the list of infected states while generating the mutants. Our goal is to be able to apply this strategy without any information on how the mutants are generated (*e.g.*, generated by other frameworks than ours) and to fairly compare with an exact approach that makes no assumption on the locality of differences. Once the set of infected states S_{infect} is obtained (by any means), the second step is to generate traces that cover such infected states.

For weak mutation (WM BS), a trace t is selected (*select* call on lines 1 and 7 in Algorithm 1) by concatenating the actions of (i) the shortest walk from the initial state i to a randomly chosen state $a \in S_{infect}$ and (ii) a random walk starting from a. To proceed, the first step during trace generation is to compute the shortest distance (*i.e.*, the number of transitions) between each state of the original TS o (or its mutant m resp.) and the initial state i of o (or m resp.) using a standard breadth-first search [28]. For strong mutation (SM BS), instead of a random walk starting from a, the algorithm will consider the actions of a path starting from a and returning to i using the computed shortest distance: the distance from a to i will (not strictly) decrease each time a transition is taken in the path.

3.3.3. Estimating the Number of Required Runs

One of the parameters for Algorithm 1 is the number of traces from the original (resp. mutant) model and run on the mutant (resp. original) model: $N/2$. There are two approaches depending whether we use random simulations or biased simulations.

Random simulations

To estimate the the number of runs, we use the method proposed by Herault *et al.* [29]. In their work, they use the Chernoff-Hoeffding bound to approximate the number of runs needed in probabilistic model checking using an approximation parameter $\epsilon > 0$ and a confidence parameter $\delta < 1$.

Under the hypothesis that traces are uniformly distributed, we can use the same method to bound the equivalence probability and estimating the number of runs needed to achieve these bounds. If $N \geq \frac{4\log(2/\delta)}{\epsilon^2}$ then we have:

$$Pr\left[equiv(m,o)\right] = Pr\left[\left|\frac{A}{N/2}\right| \leq \epsilon\right] \geq 1-\delta$$

Where A is the number of successful runs that is either $m \stackrel{t}{\Longrightarrow}$ or $o \stackrel{t}{\Longrightarrow}$ for a given trace t.

We compute $2A/N$ only when the algorithm has exhausted all the runs and set $N = \frac{8\log(2/\delta)}{\epsilon^2}$ for the number of runs as we have to account for two-way simulation: the number of runs is thus doubled.

Biased simulations

Regarding *biased simulations*, the distribution of traces will *not be uniform* as the infected states force traces to explore only given portions of the model, *viz.* where the mutations are. In this case, there is no analytical method to estimate the number of runs beforehand and N can be fixed arbitrarily. In our experiments, we simply set the same number of runs as for random simulations in order to allow comparison of the random and biased simulation algorithms.

4. Empirical Assessment

This section presents the empirical assessment of the ALE, RS, and BS approaches. We define the following research questions:

RQ1 *What is the impact of weak and strong mutation on BS/RS vs. ALE performance?*

RQ2 *How many non-equivalent mutants are effectively detected by the RS and BS approaches?*

RQ3 *What are the worst case execution times for the ALE and BS/RS approaches?*

4.1. Protocol

To answer these RQs we consider several models (from different kinds of systems), where we apply the following procedure: (i) we generate a set of mutants from the model using the operators presented in Table 2 for orders 1, 2, 5, and 10; (ii) for each order, we sample 100 non-equivalent mutants (using the ALE algorithm to guarantee non-equivalence) to form the mutant set M; (iii) for each mutant in M, we measure the execution time and result of: 3 executions of weak mutation random and biased search (WM RS/BS), and 3 executions of strong mutation-biased search (SM BS) algorithms[3] with 4 different values of δ and ϵ; and the executions of the ALE algorithm. In the following we detail the different steps of the procedure. The assessment has been performed on a Debian 3.16.7 x86_64 GNU/Linux running on a 16 cores, 2.2 GHz, 16Gb RAM virtual machine.

4.1.1. Models

We carry out the assessment on 12 different models coming from different sources and with varying size detailed in Table 1. The different characteristics considered are: the number of states (*States*); the number of transitions (*Trans.*); the number of actions (*Act.*); the number of incoming plus outgoing transitions per state (*Avg. deg.*); the maximal number of states between the

[3] As explained in section 3.3.1, SM RS is not considered for the assessment due to the poor results during our initial attempts.

Table 1: Models characteristics

Model	States	Trans.	Act.	Avg. deg.	BFS height	Back lvl tr.
S. V. Mach.	9	13	14	1.44	5	3
C. P. Term.	16	17	15	1.55	7	4
Minepump	25	41	23	4.64	15	9
Claroline	106	2,055	106	19.37	1	105
Elsa-RR	384	1,214	384	3.16	194	174
Elsa-RRN	615	1,771	615	2.88	369	289
AGE-RR	772	6,639	772	8.60	328	408
AGE-RRN	1,101	10,960	1,101	9.96	426	662
Random 1	10,000	13,652	120	1.37	7,924	3,303
Random 2	15,000	20,488	300	1.37	11,865	4,899
Random 3	15,000	20,488	210	1.37	11,865	4,899
Random 4	15,000	20,488	150	1.37	11,865	4,899

initial state and any other state when traversing the TS in breadth-first search (*BFS h.*); the number of transitions with a source state that has a higher *BFS h.* value than its destination state (*Back lvl tr.*). The models are: the soda vending machine model (*S.V.Mach.*), a small example describing the behaviour of a machine selling soda and tea [30]; the card payment terminal (*C.P.Term.*), also a small example describing the behaviour of a terminal used in a store to pay by card; the mine pump (*Minepump*), a well-known specification exemplar that models the behaviour of a pump keeping a mine safe from flooding by pumping water from a sink while avoiding methane explosions [30]; the Claroline website (*Claroline*), representing the navigational usages of a real online course management platform. The latter was reverse-engineered from an Apache log using a 2-gram inference method from Sprenkle *et al.* [31]; WordPress models (*AGE-RR*, *AGE-RRN*, *Elsa-RR*, and *Elsa-RRN*) that represent the navigational usages of two different real WordPress instances. They were also reverse-engineered using the same 2-gram inference method [31]. the *AGE-RR* and *Elsa-RR*, we considered only request type (*e.g.*, POST, GET, HEAD) and the requested resource (*e.g.*, "/index.php") in the sequences used. For the *AGE-RRN* and *Elsa-RRN* models, we considered request type, requested resource, and parameter names (*e.g.*, "?page=") in the sequences used as input of the 2-gram inference method.

The random models (*Random 1* to *Random 4*) were generated according to the following procedure: (i) generate a set of random oriented graphs and compute the different measures from Table 1 (except number of actions); (ii) select those graphs that are likely to represent a real system according to Pelánek [32], *i.e.*, those having a small average degree, a large BFS height and a small number of back level edges (in this order); (iii) apply a random labelling multiple times and compute the occurrence probability, *i.e.*, the probability of the labels to obtain a set of randomly generated TSs; (iv) select the TS that has

Table 2: Transition system mutation operators

SMI	State Missing operator removes a state (other than the initial state) and all its incoming/outgoing transitions.
WIS	Wrong Initial State operator changes the initial state.
AEX	Action Exchange operator replaces the action linked to a given transition by another action.
AMI	Action Missing operator removes an action from a transition.
TMI	Transition Missing operator removes a transition.
TAD	Transition Add operator adds a transition between two states.
TDE	Transition Destination Exchange operator modifies the destination of a transition.

the following properties[4]: the probability of the most frequently occurring label in the TS is less than, or equal to, 6%, and the cumulated probability of the 5 most frequently occurring labels is less than or equal to 20% [33]. We end up with 4 random models as recorded in Table 1.

4.1.2. Mutant Generation and Sampling

First-order mutants are generated using the operators presented in Table 2. Each operator is applied (arbitrarily) 10 times on the *S.V.Mach.*, *C.P.Term.*, and *Minepump* models. Due to the small size of the models, applying the same mutation operator more than 10 times is not relevant. Operators are also applied (arbitrarily) 500 times on the other models. In the same way, N-order mutants (with N equal to 2, 5, or 10 in our case) are generated by applying the same operators 10 or 500 times (depending on the model) on $(N-1)$-order mutants. After the generation, we perform a random sampling of 100 mutants (when available) for orders 1, 2, 5, and 10, giving us a set M with 370 mutants for the *S.V.Mach.*, *C.P.Term.*, and *Minepump* models, and 400 mutants for the other models. To ease mutant generation, we use our compact representation [34].

4.1.3. Non-determinism

We checked all the 4,710 mutants and found that only 3.54% of them are non-deterministic. Nevertheless, there is a great disparity amongst models as the non-determinism rate varies from 0% for *Elsa-RRN* to 15.5% for *Claroline*. Higher-order mutation greatly influenced non-determinism rates: the sole order 10 is responsible for 53% of all non-deterministic mutants. In terms of mutation operators, TAD accounts for a large majority of non-deterministic first-order mutants (78%) and AEX for the remaining 22%. At higher orders, these two operators are largely involved. They are absent only in the *Minepump* model where TDE and AMI appear for two non-deterministic mutants.

[4]These properties are likely to represent real systems [32]

4.1.4. Algorithm Execution

To run the language equivalence algorithms (for WM and SM), we use the HKC library [35], an OCaml implementation of the ALE algorithm [17] compiled using OCamlbuild. This tool handles non-deterministic TSs using different strategies: the automata may be processed either forward of backwards, and the exploration strategy may be breadth-first or depth-first. For each mutant, we execute the HKC library using each of the 4 possible configurations. The input models and their mutants have been transformed from our XML format to the Timbuk input format supported by HKC.

The random and biased simulation algorithms are implemented in Java using multi-threading to parallelize trace selection and execution as described in Algorithm 1 (lines 1, 3, 7, and 9). In our experiments, we set up the algorithm with 4 threads and run 4 instances in parallel on our virtual machine with 16 cores. We run the simulation algorithms with 4 different values of δ and ϵ determining the number of traces selected and executed (N in Algorithm 1):

- RS1/BS1: ($\delta = 1e-10$, $\epsilon = 0.01$, $N = 1,897,519$);
- RS2/BS2: ($\delta = 1e-10$, $\epsilon = 0.1$, $N = 18,975$);
- RS3/BS3: ($\delta = 1e-5$, $\epsilon = 0.1$, $N = 9,764$);
- RS4/BS4: ($\delta = 1e-1$, $\epsilon = 0.1$, $N = 2,396$).

For all the simulation configurations and all models, we fixed the trace length k to 3,000, which was our compromise between performance and non-equivalence detection: setting k to BFS height led to crashes in some cases. In order to answer RQ3, we also run each algorithm (RS1/BS1 to RS4/BS4, plus the 4 possible ALE configurations) with the model itself as the "mutant". Those (unrealistic) equivalent detection runs between the model and itself are only used to approximate the worst computation time of the different algorithms.

4.2. Results and Discussion

4.2.1. Random/biased simulations and ALE - Answering RQ1

Figure 1 presents the execution time per mutant of the studied algorithms, which is detailed in the Appendix. Regarding weak mutation scenarios, the ALE approach is the fastest in eleven of our models. On the *AGE-RN* model, biased simulations are faster for the largest numbers of runs. However, the results are at the limit of non-significance (see Table 3), so that the only clearly significant result is for *BS1* on this model. For *AGE-RNN*, execution times for biased simulations are non-significant. Random simulations are also faster than ALE on *AGE-RRN* but only certain settings are significant. We thus conclude that the ALE approach is more interesting in terms of execution time. When we compare the two forms of simulations, for the smallest models, biased simulations are either on par for the smallest models or slightly better. Additional computations such as the breath-first search used for biased simulation do not cause significant overhead. For the largest random models, random simulations are faster. In

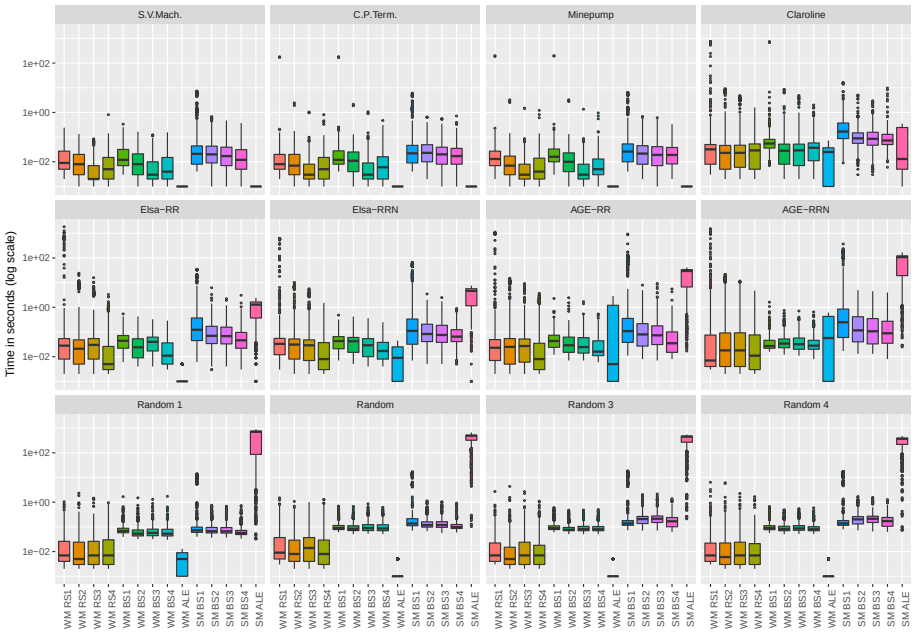

Figure 1: Execution time of the equivalent mutant detection

these cases, the overhead of computing infected states and paths that cover these states is greater and random simulation is faster. However, lower standard deviations for biased simulation execution times over random ones make the BS approach easier to use.

Regarding strong mutation, several observations can be made. First, random simulations provide very high execution times compared to biased simulations or the ALE algorithm (the analysis of one model is stopped after one hour). This may be due to the difficulty to reach the initial state again when performing random walks in the TSs. Second, biased simulations are faster than ALE executions for models larger than 300 states. On the largest models, biased simulations can be up to 1,000 times faster. We thus conclude that these are the most interesting situations in which to use BS, for mutation analysis. On smaller models, the ALE algorithm's performance is quite impressive and therefore should be privileged.

4.2.2. Non-equivalent mutant detection - Answering RQ2

To answer RQ2, we compute the non-equivalent mutant classification *recall* of the BS/RS algorithms (in Figure 2): *i.e.*, the percentage of non-equivalent mutants detected by the BS/RS amongst the selected mutants. By construction, the ALE algorithm has a recall of 100%, it is therefore not shown here. It is also noted that the precision is 100% since all the non-equivalent mutants detected

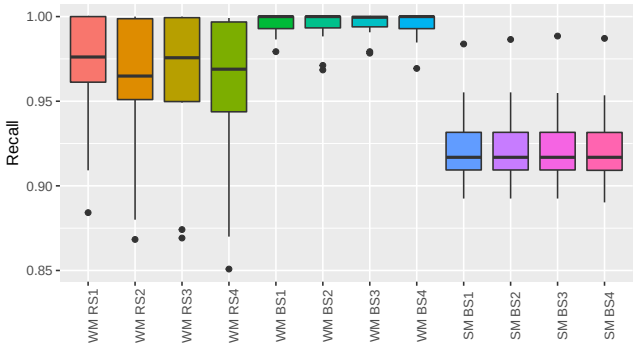

Figure 2: Percentage of non-equivalent mutants correctly detected (recall)

are indeed killable, by construction of our mutant set.

All our simulations detect more than 85% of the non-equivalent mutants, with a clear advantage for biased simulations which never achieve worse than 95% for the weak mutation scenario. As for time, deviation in the recall is smaller for biased simulations thus making the approach more predictable in addition of being more reliable. We also observe that the random simulations are more sensitive to the number of runs: we need more of them to discover discrepancies by luck. This effect cannot be observed for biased simulations. A possible explanation is that the number of runs required to cover infected states with traces is lower than the number we provided.

For strong mutation, the BS approach's recall decreases to around 92% ($\overline{recall} = 92\%$, $\sigma = 3\%$): amongst the 5,113 non-equivalent mutant non-detections (over a total of 64,529 non-equivalent mutant evaluations), 1,905 (37%) were TAD mutants, 1,755 (34%) were WIS mutants, 545 (11%) were TDE mutants, and 459 (9%) were 2nd-order TAD mutants (*i.e.*, TAD-TAD mutants); the rest of non-equivalent mutants not detected is distributed amongst different operators with less than 2% for each. This decrease may be due to the difficulty to find a path to the initial state: for strong mutation, the BS trace selection algorithm will consider traces starting from, and ending in, the initial state. This means that mutations creating (TAD) or modifying (TDE) a back-level transition will not be detected using SM BS. Concerning WIS mutants, we believe that, as the WIS operator only changes the initial state of the TS, the set of infected states (S_{infect}) is empty, which is equivalent in our implementation of SM BS to considering all the states infected.

4.2.3. Worst case scenario (execution time) - Answering RQ3

Figure 3 presents a compact view of the worst execution time of the different algorithms (RQ3). We grouped the different results by the kind of model: embedded system, web-application, or randomly generated model. As expected, the RS/BS execution time is directly correlated to the δ and ϵ values: a lower

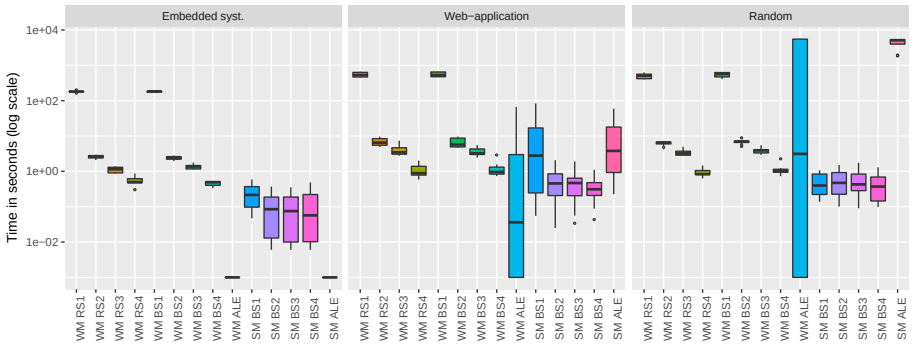

Figure 3: Worst execution time of the equivalent mutant detection using the model itself as mutant

number of traces selected and executed (N) takes less time. Overall, the time of the ALE executions grows with the size of the model, reaching 5,660 seconds (more than one and a half hour) for the worst WM ALE execution time on the Random 2 model.

4.3. Threats to Validity

4.3.1. Internal Validity

We performed our experiment on 12 models: 3 academic examples (*S. V. Mach.*, *C. P. Term.*, *Minepump*), 5 larger real-world models (*Claroline*, *Elsa-RR*, *Elsa-RRN*, *AGE-RR*, and *AGE-RRN*) and 4 randomly generated models (*Random 1-4*). These models come from different sources and represent two different kinds of systems: embedded systems designed by an engineer and web-based applications where the model has been reverse-engineered from a running instance using a 2-gram inference method [31]. The random models were built from a set of generated TSs in order to match the real system state-space measures, as described by Pelánek [32, 33].

4.3.2. Construct Validity

The RS/BS δ and ϵ values have been arbitrarily chosen. The first values (RS1/BS1: $\delta = 1e-10$, $\epsilon = 0.01$) are the same as in Hérault *et al.* [29]. As the number of traces selected and executed N equals to $\frac{8\log(2/\delta)}{\epsilon^2}$, we chose to run the algorithm with 3 higher parameters values in order to reduce N. We cannot guarantee that our parameter values are relevant for any model. They will rather depend on the model size, the desired approximation (ϵ) and confidence (δ), and the time budget allowed for the equivalence analysis.

To the best of our knowledge, the HKC library [35] was the only publicly available tool able to perform ALE checking on non-deterministic TSs. We cannot guarantee that there are no other other tools providing the same features

Table 3: P-values of the Wilcoxon rank sum test between the WM RS/BS execution times and the WM ALE execution times.

Model	**WM RS1**	**WM RS2**	**WM RS3**	**WM RS4**
S.V.Mach.	$\leq 2.2e-16$	$\leq 2.2e-16$	$\leq 2.2e-16$	$\leq 2.2e-16$
C.P.Term.	$\leq 2.2e-16$	$\leq 2.2e-16$	$\leq 2.2e-16$	$\leq 2.2e-16$
Minepump	$\leq 2.2e-16$	$\leq 2.2e-16$	$\leq 2.2e-16$	$\leq 2.2e-16$
Claroline	$\leq 2.2e-16$	$\leq 2.2e-16$	$\leq 2.2e-16$	$\leq 2.2e-16$
Elsa-RR	$\leq 2.2e-16$	$\leq 2.2e-16$	$\leq 2.2e-16$	$\leq 2.2e-16$
Elsa-RRN	$\leq 2.2e-16$	$\leq 2.2e-16$	$\leq 2.2e-16$	$\leq 2.2e-16$
AGE-RR	$2.866e-03$	$9.676e-03$	$2.021e-02$	$\mathbf{3.249e-01}$
AGE-RRN	$\mathbf{8.143e-02}$	$8.379e-04$	$6.981e-04$	$2.162e-02$
Random 1	$\leq 2.2e-16$	$\leq 2.2e-16$	$\leq 2.2e-16$	$\leq 2.2e-16$
Random 2	$\leq 2.2e-16$	$\leq 2.2e-16$	$\leq 2.2e-16$	$\leq 2.2e-16$
Random 3	$\leq 2.2e-16$	$\leq 2.2e-16$	$\leq 2.2e-16$	$\leq 2.2e-16$
Random 4	$\leq 2.2e-16$	$\leq 2.2e-16$	$\leq 2.2e-16$	$\leq 2.2e-16$
	WM BS1	**WM BS2**	**WM BS3**	**WM BS4**
S.V.Mach.	$\leq 2.2e-16$	$\leq 2.2e-16$	$\leq 2.2e-16$	$\leq 2.2e-16$
C.P.Term.	$\leq 2.2e-16$	$\leq 2.2e-16$	$\leq 2.2e-16$	$\leq 2.2e-16$
Minepump	$\leq 2.2e-16$	$\leq 2.2e-16$	$\leq 2.2e-16$	$\leq 2.2e-16$
Claroline	$\leq 2.2e-16$	$\leq 2.2e-16$	$\leq 2.2e-16$	$\leq 2.2e-16$
Elsa-RR	$\leq 2.2e-16$	$\leq 2.2e-16$	$\leq 2.2e-16$	$\leq 2.2e-16$
Elsa-RRN	$\leq 2.2e-16$	$\leq 2.2e-16$	$\leq 2.2e-16$	$\leq 2.2e-16$
AGE-RR	$9.107e-03$	$4.744e-02$	$\mathbf{6.405e-02}$	$\mathbf{1.382e-01}$
AGE-RRN	$\mathbf{5.991e-01}$	$\mathbf{7.076e-01}$	$\mathbf{5.674e-01}$	$\mathbf{5.168e-01}$
Random 1	$\leq 2.2e-16$	$\leq 2.2e-16$	$\leq 2.2e-16$	$\leq 2.2e-16$
Random 2	$\leq 2.2e-16$	$\leq 2.2e-16$	$\leq 2.2e-16$	$\leq 2.2e-16$
Random 3	$\leq 2.2e-16$	$\leq 2.2e-16$	$\leq 2.2e-16$	$\leq 2.2e-16$
Random 4	$\leq 2.2e-16$	$\leq 2.2e-16$	$\leq 2.2e-16$	$\leq 2.2e-16$

with lower execution time. To avoid bias in the random selections in the RS/BS algorithms, we execute each configuration of the different algorithms 3 times.

4.3.3. External Validity

We cannot guarantee that our results are generalizable to all behavioural models. However, we recall the diversity of the model sources (hand-crafted, reverse-engineered, and randomly generated to match real system state-space) as well as the diversity of the considered systems. Variations in performance of the algorithms also suggest mitigation of this threat.

4.3.4. Conclusion Validity

To confirm our observations on the recall of the RS/BS algorithms, we test the null hypothesis between the outputs of our algorithm (the mutant is equivalent/non-equivalent) and a random equivalent/non-equivalent assignment using a Wilcoxon rank sum test. The p-value lower than 2.2e-16[5] discredits the null hypothesis showing that the equivalent/non-equivalent detection recall is significant.

To confirm the statistical difference between the execution times of the RS/BS and ALE algorithms, we test the null hypothesis between RS/BS execution time and ALE execution time for weak and strong mutation for each of our input models using a Wilcoxon rank sum test. For weak mutation, the

[5]Value $2.2e-16$ corresponds to the smallest possible p-value computable with R.

results of this statistical test are shown in Table 3: for every model except *AGE-RR*/*AGE-RRN* models, the p-value is lower than 2.2e-16, discrediting the null hypothesis and showing a significant difference in the execution times. The execution times of *AGE-RR*/*AGE-RRN* model are only significant for RS1 to RS3, BS1, and BS3 (for *AGE-RR*); and RS2 to RS4 (for *AGE-RRN*). For strong mutation, all the p-values were lower than 2.2e-16, showing a significant difference in execution time between the BS algorithm and the ALE algorithm in a strong mutation scenario.

4.3.5. Verifiability

The input models, as well as the tools and scripts used to perform the empirical assessment, are available online at `https://projects.info.unamur.be/vibes/mutants-equiv.html`. The input models are encoded using an XML format and are processed by our Java tools (part of VIBeS [36]) for the RS/BS algorithms. The ALE execution is done using HKC [35]. Both VIBeS and HKC are released under open source licences (MIT license for VIBeS, GNU LGPL for HKC), allowing one to inspect, reuse, or adapt the code. VIBeS's source code is available online in a GitHub repository at `https://github.com/xdevroey/vibes`, and the different Maven artefacts were deployed on the Maven central repository. As our assessment involves randomization, the complete results are also downloadable as well as the script files used to perform the analysis described in section 4.2. Finally, one may (re-)run the complete assessment using the provided Makefile.

4.4. Lessons Learned

From our experiment we draw the following lessons. (i) Regarding weak mutation and independently of the size or nature of the models, the ALE approach provides faster and exact answers. This indicates that state-of-the-art language equivalence algorithms can be used successfully for such a task. (ii) Regarding strong mutation, biased random simulations are of interest for the web and the random models, and gains increase with the size (from one to three orders of magnitude). Having simulations with recall values above 90% allow their use as reasonably reliable fast filters that can discard non-equivalent mutants, leaving to ALE algorithms the difficult cases. This helps accelerating the analysis of large number of mutants. (iii) Biased simulations are more predictable in terms of execution time and recall. Additionally, drastically increasing the number of runs does not affect their performance as opposed to random simulations. (iv) The configuration of the ALE algorithm (forward/backward processing, or breadth-first or depth-first exploration) has little influence on the total execution time (regarding equivalent mutant detection). This may be explained by the fact that mutations occur randomly and therefore do not privilege any graph traversal strategy.

5. Related Work

The usage of simulation heuristics for testing purposes is presented in Section 5.1). Approaches related to the equivalent mutant problem and model-based mutation are then discussed in Sections 5.2 and 5.3, respectively.

5.1. Simulation

Our random simulation heuristic, which yields a probabilistic interpretation of the problems under analysis by making several repeated samples, is akin to Monte-Carlo simulation. Monte-Carlo methods were found to be quite efficient for searching and reasoning on large data spaces. In software verification, Monte Carlo simulations have been used to devise statistical model-checking techniques [37, 29] that alleviate state explosion. In software testing, Langdon *et al.* [38] used them, together with genetic programming, in order to identify subsuming higher-order mutants. Poulding and Feldt [39] used a variant of the method, called Nested Monte-Carlo Search, to generate random data structures to be used for testing. Along the same lines, Nested Monte-Carlo Search was used, by Poulding and Feldt [40] to heuristically perform model checking of Java programs. All these methods are related to ours since they use Monte-Carlo. However, none of them aims at modelling mutants or tackling the equivalent mutant problem.

Walkinshaw and Bogdanov [41] argue that using random selection (like Lo and Khoo [42]) in order to compare automata languages may be biased due to the impossibility to obtain a representative sample of the language. In their work, they use a model-based testing approach (the W-method [43]) to compare two automata from the accepted language perspective, and a *diff* algorithm to compare them with respect to their transition structures (which is a more elaborate version of our heuristic used to compute the set of infected states S_{infect}). In contrast, we look for difference instead of similarity, which motivates the choice of easier-to-compute random heuristics as baselines to compare with an ALE approach.

5.2. Equivalent mutants

Previous work demonstrated that equivalent mutants skew the mutation score measurements and thus hinder the effectiveness of the method [44, 4]. Unfortunately, it has been proven that judging whether a code mutant is equivalent to the original code is an undecidable problem [45]. This means that there is no solution to the general case of this problem. Luckily, since mutations are small syntactic changes, heuristics can identify several classes of them [15]. Two types of such heuristics exist in the literature: those that operate in a *static* manner and those that are *dynamic*.

Static techniques include the use of compiler optimizations [46], constraint solving [25], program slicing [47], data-flow patterns [48], and formal verification [26]. All these techniques are effective at detecting certain types of equivalent mutants, *i.e.*, trivial equivalencies [15], but unfortunately, they are not applicable to model mutants.

Dynamic techniques measure the differences between the test executions of the original and mutant programs and identify likely non-equivalent mutants. Schuler and Zeller [49] and Papadakis *et al.* [50] measure the impact on coverage, while Kintis *et al.* [51] measure the impact on other mutants (second-order mutants). Our technique shares the same notion of equivalence because we check the model trace in order to judge it. However, we do not consider executable code as we only deal with model mutants. We also sample execution in order to increase the efficiency of the process. It is to be noted that we have a different notion of equivalence since we deal with behavioural models. Therefore, differences in traces imply different behaviours, which is not the case for executable code.

Non-determinism complicates equivalence detection both at the code [52] and model levels [53]. Patel and Hierons [52] associate predictions from pairs of inputs and outputs of the mutant program and check whether these predictions can be discarded by the original program, hence showing non-equivalence. This is not applicable to our case since our models do not have outputs. Aichernig and Jöbstl [53] also encode the semantics of the action models in terms of constraints and use refinement to check conformance in the context of non-determinism. In our case, RS/BS manage non determinism in the TSs by considering all the possible runs.

Perhaps the closest work is that of Papadakis and Malevris [54] who sample execution paths according to their length (select the k-shortest paths), symbolically execute them and judge mutant equivalence based on the selected paths. The main differences with our approach are that we additionally sample paths that cover infected states and we operate on behavioural models instead of actual code representation.

5.3. Model-based mutation

Specification mutation testing aims at identifying defects on the implementations under test by altering the models of the system and requiring the design of tests that identify these differences [11]. The main point about this technique is that it complements code-based testing by targeting problems related to missing functionality [13, 12].

Given the plethora of the existing models and languages, many model-based mutation techniques have been developed. Woodward [55], Fabbri *et al.* [56] and Hierons and Merayo [57] suggested a set of mutant operators for algebraic specifications, finite state machines and Statecharts, and probabilistic finite state machines, respectively. Similarly, Henard *et al.* [58], Arcaini *et al.* [59] and Papadakis *et al.* [10] mutated feature models and combinatorial interaction models.

Regarding behavioural models, like the ones we used here, Aichernig *et al.* [21, 60] developed a mutation-based test generation technique for state machines. Belli and Beyazit [61] compare mutation-testing strategies when applied on event-based and state-based models, and found that both had similar effectiveness. In follow-up studies, Belli *et al.* [62] and Aichernig *et al.* [14] evaluated

their model-based mutation testing approaches on industrial systems and found that they were complementary, in terms of fault detection, to code-based testing.

Generally, the EMP is seldom the single focus of the above approaches as it is in the present study.

6. Conclusion

In this paper, we investigated the relevance of an exact language equivalence approach to tackle the equivalent mutant problem at the model level. To do so, we offered two algorithms based on random simulation and compared them to language equivalence under the weak and the strong mutation testing scenarios. Our experiments demonstrated the efficiency of the exact approach for the weak mutation case. For the strong mutation case, our biased simulations – that pre-process the models to detect states that are infected by mutations – are efficient and up to 1,000 times faster on models that contain more than 300 states. Though, our simulations introduce detection errors up to 8%. These results suggest that the equivalence analysis can be performed much faster by using the simulations, to quickly discard many non-equivalent mutants, and employing exact approaches only on the small number of probably equivalent mutants that remain.

There is room for improvement. First, we will extend our experiments to other forms of equivalence and tools. We would also like to switch from the pure equivalence analysis to test generation concerns by analysing counter-examples. Our long-term goal is to draw attention on the applications of language equivalence for mutation testing and develop further EMP-dedicated solutions.

Acknowledgements

We would like to thank Damien Pous for his support on the HKC tool. This research was partially funded by the EU Project STAMP ICT-16-10 No.731529 and the Dutch 4TU project "Big Software on the Run" as well as by the European Regional Development Fund (ERDF) "Ideas for the future Internet" (IDEES) project.

References

[1] X. Devroey, G. Perrouin, M. Papadakis, A. Legay, P.-Y. Schobbens, P. Heymans, Automata Language Equivalence vs. Simulations for Model-Based Mutant Equivalence: An Empirical Evaluation, in: 2017 IEEE International Conference on Software Testing, Verification and Validation (ICST), IEEE, Tokyo, Japan, 2017, pp. 424–429. doi:10.1109/ICST.2017.46.

[2] J. H. Andrews, L. C. Briand, Y. Labiche, A. S. Namin, Using Mutation Analysis for Assessing and Comparing Testing Coverage Criteria, IEEE Transactions on Software Engineering 32 (8) (2006) 608–624. doi:10.1109/TSE.2006.83.

[3] J. Offutt, A mutation carol: Past, present and future, Information and Software Technology 53 (10) (2011) 1098–1107. doi:10.1016/j.infsof.2011.03.007.

[4] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. L. Traon, M. Harman, Mutation testing advances: An analysis and survey, Advances in Computers 112.

[5] M. Papadakis, N. Malevris, Automatic mutation test case generation via dynamic symbolic execution, in: International Symposium on Software Reliability Engineering, ISSRE, IEEE, 2010, pp. 121–130. doi:10.1109/ISSRE.2010.38.

[6] G. Fraser, A. Arcuri, Achieving scalable mutation-based generation of whole test suites, Empirical Software Engineering (2014) 1–30doi:10.1007/s10664-013-9299-z.

[7] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, G. Fraser, Are Mutants a Valid Substitute for Real Faults in Software Testing?, in: International Symposium on the Foundations of Software Engineering, FSE, ACM, 2014, pp. 654–665. doi:10.1145/2635868.2635929.

[8] T. T. Chekam, M. Papadakis, Y. L. Traon, M. Harman, An empirical study on mutation, statement and branch coverage fault revelation that avoids the unreliable clean program assumption, in: Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017, 2017, pp. 597–608. doi:10.1109/ICSE.2017.61.

[9] R. Baker, I. Habli, An empirical evaluation of mutation testing for improving the test quality of safety-critical software, IEEE Transactions on Software Engineering 39 (6) (2013) 787–805. doi:10.1109/TSE.2012.56.

[10] M. Papadakis, C. Henard, Y. Le Traon, Sampling program inputs with mutation analysis: Going beyond combinatorial interaction testing, in: International Conference on Software Testing, Verification and Validation, ICST, IEEE, 2014, pp. 1–10. doi:10.1109/ICST.2014.11.

[11] T. A. Budd, A. S. Gopal, Program testing by specification mutation, Computer Languages 10 (1) (1985) 63–73. doi:10.1016/0096-0551(85)90011-6.

[12] W. E. Howden, Reliability of the path analysis testing strategy., IEEE Transactions on Software Engineering 2 (3) (1976) 208–215. doi:10.1109/TSE.1976.233816.

[13] J. M. Voas, G. McGraw, Software Fault Injection: Inoculating Programs Against Errors, John Wiley & Sons, Inc., 1997. doi:10.1002/(SICI)1099-1689(199903)9:1<75::AID-STVR174>3.0.CO;2-T.

[14] B. K. Aichernig, J. Auer, E. Jöbstl, R. Korosec, W. Krenn, R. Schlick, B. V. Schmidt, Model-based mutation testing of an industrial measurement device, in: Tests and Proofs, Vol. 8570 of LNCS, Springer, 2014, pp. 1–19. doi:10.1007/978-3-319-09099-3_1.

[15] M. Papadakis, Y. Jia, M. Harman, Y. Le Traon, Trivial compiler equivalence: A large scale empirical study of a simple fast and effective equivalent mutant detection technique, in: International Conference on Software Engineering, ICSE, IEEE, 2015, pp. 936–946. doi:10.1109/ICSE.2015.103.

[16] O. Kupferman, M. Y. Vardi, Verification of fair transition systems, in: Computer Aided Verification, Springer, 1996, pp. 372–382. doi:10.1007/3-540-61474-5_84.

[17] F. Bonchi, D. Pous, Checking NFA equivalence with bisimulations up to congruence, in: Symposium on Principles of Programming Languages, POPL, ACM, Rome, Italy, 2013, pp. 457–468. doi:10.1145/2429069.2429124.

[18] L. Doyen, J. Raskin, Antichain algorithms for finite automata, in: Tools and Algorithms for the Construction and Analysis of Systems, TACAS, Vol. 6015 of LNCS, Springer, 2010, pp. 2–22. doi:10.1007/978-3-642-12002-2_2.

[19] P. Ammann, J. Offutt, Introduction to software testing, Cambridge University Press, 2008.

[20] C. Baier, J. Katoen, Principles of model checking, MIT Press, 2008.

[21] B. K. Aichernig, E. Jöbstl, S. Tiran, Model-based mutation testing via symbolic refinement checking, Science of Computer Programming 97 (2015) 383–404. doi:10.1016/j.scico.2014.05.004.

[22] E. Jöbstl, Model-based mutation testing with constraint and smt solvers, Ph.D. thesis, Graz University of Technology (2014).

[23] R. A. DeMillo, A. J. Offutt, Experimental results from an automatic test case generator, ACM Transactions on Software Engineering and Methodology (TOSEM) 2 (2) (1993) 109–127. doi:10.1145/151257.151258.

[24] W. E. Howden, Weak mutation testing and completeness of test sets, IEEE Transactions on Software Engineering SE-8 (4) (1982) 371–379. doi:10.1109/TSE.1982.235571.

[25] A. J. Offutt, J. Pan, Automatically detecting equivalent mutants and infeasible paths, Software Testing, Verification and Reliability 7 (3) (1997) 165–192. doi:10.1002/(SICI)1099-1689(199709)7:3<165::AID-STVR143>3.0.CO;2-U.

[26] S. Bardin, M. Delahaye, R. David, N. Kosmatov, M. Papadakis, Y. Le Traon, J. Marion, Sound and quasi-complete detection of infeasible test requirements, in: International Conference on Software Testing, Verification and Validation, ICST, IEEE, Graz, Austria, 2015, pp. 1–10. doi:10.1109/ICST.2015.7102607.

[27] W. Krenn, R. Schlick, Mutation-driven test case generation using short-lived concurrent mutants - first results, CoRR abs/1601.06974.

[28] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, Introduction to algorithms, Vol. 6, MIT press Cambridge, 2001.

[29] T. Hérault, R. Lassaigne, F. Magniette, S. Peyronnet, Approximate probabilistic model checking, in: Verification, Model Checking, and Abstract Interpretation, 5th International Conference, VMCAI, Vol. 2937 of LNCS, Springer, Venice, Italy, 2004, pp. 73–84. doi:10.1007/978-3-540-24622-0_8.

[30] A. Classen, Modelling with FTS: a Collection of Illustrative Examples, Tech. Rep. P-CS-TR SPLMC-00000001, PReCISE Research Center, University of Namur, Namur, Belgium (2010).
URL https://projects.info.unamur.be/fts/publications/

[31] S. E. Sprenkle, L. L. Pollock, L. M. Simko, Configuring effective navigation models and abstract test cases for web applications by analysing user behaviour, Software Testing, Verification and Reliability 23 (6) (2013) 439–464. doi:10.1002/stvr.1496.

[32] R. Pelánek, Typical Structural Properties of State Spaces, in: International SPIN Workshop, Vol. 2989 of LNCS, Springer, 2004, pp. 5–22. doi:10.1007/978-3-540-24732-6_2.

[33] R. Pelánek, Properties of state spaces and their applications, International Journal on Software Tools for Technology Transfer 10 (5) (2008) 443–454. doi:10.1007/s10009-008-0070-5.

[34] X. Devroey, G. Perrouin, M. Papadakis, P.-Y. Schobbens, P. Heymans, Featured Model-based Mutation Analysis, in: International Conference on Software Engineering, ICSE, ACM, Austin, TX, USA, 2016. doi:10.1145/2884781.2884821.

[35] F. Bonchi, D. Pous, HKC Library v. 1.0, https://perso.ens-lyon.fr/damien.pous/hknt/ (2013).

[36] X. Devroey, G. Perrouin, Variability Intensive system Behavioural teSting (VIBeS) v. 1.1.4, https://projects.info.unamur.be/vibes/ (2015).

[37] H. L. S. Younes, R. G. Simmons, Probabilistic Verification of Discrete Event Systems Using Acceptance Sampling, in: International Conference on Computer Aided Verification, CAV, Vol. 2404 of LNCS, Springer-Verlag, London, UK, 2002, pp. 223–235. doi:10.1007/3-540-45657-0_17.

[38] W. B. Langdon, M. Harman, Y. Jia, Efficient multi-objective higher order mutation testing with genetic programming, Journal of Systems and Software 83 (12) (2010) 2416–2430. doi:10.1016/j.jss.2010.07.027.

[39] S. M. Poulding, R. Feldt, Generating structured test data with specific properties using nested monte-carlo search, in: Genetic and Evolutionary Computation Conference, GECCO, ACM, Vancouver, BC, Canada, 2014, pp. 1279–1286. doi:10.1145/2576768.2598339.

[40] S. M. Poulding, R. Feldt, Heuristic model checking using a monte-carlo tree search algorithm, in: Proceedings of the Genetic and Evolutionary Computation Conference, GECCO, ACM, Madrid, Spain, 2015, pp. 1359–1366. doi:10.1145/2739480.2754767.

[41] N. Walkinshaw, K. Bogdanov, Automated Comparison of State-Based Software Models in Terms of Their Language and Structure, ACM Transactions on Software Engineering and Methodology 22 (2) (2013) 1–37. doi:10.1145/2430545.2430549.

[42] D. Lo, S. c. Khoo, QUARK: Empirical Assessment of Automaton-based Specification Miners, in: 13th Working Conference on Reverse Engineering, 2006, pp. 51–60. doi:10.1109/WCRE.2006.47.

[43] A. P. Mathur, Foundations of software testing, Pearson Education, 2008.

[44] L. Madeyski, W. Orzeszyna, R. Torkar, M. Jozala, Overcoming the Equivalent Mutant Problem: A Systematic Literature Review and a Comparative Experiment of Second Order Mutation, IEEE Transactions on Software Engineering 40 (1) (2014) 23–42. doi:10.1109/TSE.2013.44.

[45] T. A. Budd, D. Angluin, Two Notions of Correctness and Their Relation to Testing, Acta Informatica 18 (1) (1982) 31–45. doi:10.1007/BF00625279.

[46] A. J. Offutt, W. M. Craft, Using compiler optimization techniques to detect equivalent mutants, Software Testing, Verification and Reliability 4 (3) (1994) 131–154. doi:10.1002/stvr.4370040303.

[47] R. M. Hierons, M. Harman, S. Danicic, Using program slicing to assist in the detection of equivalent mutants, Software Testing, Verification and Reliability 9 (4) (1999) 233–262. doi:10.1002/(SICI)1099-1689(199912)9:4<233::AID-STVR191>3.0.CO;2-3.

[48] M. Kintis, N. Malevris, MEDIC: A static analysis framework for equivalent mutant identification, Information & Software Technology 68 (2015) 1–17. doi:10.1016/j.infsof.2015.07.009.

[49] D. Schuler, A. Zeller, Covering and uncovering equivalent mutants, Software Testing, Verification and Reliability 23 (5) (2013) 353–374. doi:10.1002/stvr.1473.

[50] M. Papadakis, M. E. Delamaro, Y. Le Traon, Mitigating the effects of equivalent mutants with mutant classification strategies, Science of Computer Programming 95 (2014) 298–319. doi:10.1016/j.scico.2014.05.012.

[51] M. Kintis, M. Papadakis, N. Malevris, Employing second-order mutation for isolating first-order equivalent mutants, Software Testing, Verification and Reliability 25 (5-7) (2015) 508–535. doi:10.1002/stvr.1529.

[52] K. Patel, R. M. Hierons, Resolving the equivalent mutant problem in the presence of non-determinism and conicidental correctness, in: 28th IFIP International Conference on Testing Software and Systems (ICTSS), 2016. doi:10.1007/978-3-319-47443-4_8.

[53] B. K. Aichernig, E. Jobstl, Towards symbolic model-based mutation testing: Pitfalls in expressing semantics as constraints, in: 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation, 2012, pp. 752–757. doi:10.1109/ICST.2012.169.

[54] M. Papadakis, N. Malevris, Mutation based test case generation via a path selection strategy, Information & Software Technology 54 (9) (2012) 915–932. doi:10.1016/j.infsof.2012.02.004.

[55] M. R. Woodward, Errors in algebraic specifications and an experimental mutation testing tool, Software Engineering Journal 8 (4) (1993) 221–224. doi:10.1002/scj.4690271006.

[56] S. Fabbri, J. C. Maldonado, T. Sugeta, P. C. Masiero, Mutation testing applied to validate specifications based on statecharts, in: International Symposium on Software Reliability Engineering, ISSRE, IEEE, 1999, pp. 210–219. doi:10.1109/ISSRE.1999.809326.

[57] R. M. Hierons, M. G. Merayo, Mutation testing from probabilistic and stochastic finite state machines, Journal of Systems and Software 82 (11) (2009) 1804–1818. doi:10.1016/j.jss.2009.06.030.

[58] C. Henard, M. Papadakis, G. Perrouin, J. Klein, Y. Le Traon, Assessing Software Product Line Testing Via Model-Based Mutation: An Application to Similarity Testing, in: International Conference on Software Testing, Verification and Validation Workshops, ICSTW, IEEE, Luxembourg, Luxembourg, 2013, pp. 188–197. doi:10.1109/ICSTW.2013.30.

[59] P. Arcaini, A. Gargantini, P. Vavassori, Generating tests for detecting faults in feature models, in: International Conference on Software Testing, Verification and Validation, ICST, IEEE, Graz, Austria, 2015, pp. 1–10. doi:10.1109/ICST.2015.7102591.

[60] W. Krenn, R. Schlick, S. Tiran, B. K. Aichernig, E. Jöbstl, H. Brandl, Momut: : UML model-based mutation testing for UML, in: International Conference on Software Testing, Verification and Validation, ICST, IEEE, Graz, Austria, 2015, pp. 1–8. doi:10.1109/ICST.2015.7102627.

[61] F. Belli, M. Beyazit, Event-Based Mutation Testing vs. State-Based Mutation Testing - An Experimental Comparison, in: International Conference on Computers, Software & Applications, COMPSAC, IEEE, 2011, pp. 650–655. doi:10.1109/COMPSAC.2011.90.

[62] F. Belli, C. J. Budnik, A. Hollmann, T. Tuglular, W. E. Wong, Model-based mutation testing - approach and case studies, Science of Computer Programming 120 (2016) 25–48. doi:10.1016/j.scico.2016.01.003.

Appendix A. Complete execution results

This appendix presents the results of the different weak and strong mutation ALEs/BSs/RSs algorithms. For each algorithm, a table gives the execution parameter values (δ, ϵ, and the resulting number of runs **N**), the recall, the average execution time ($\overline{\textbf{time}}$), and the standard deviation (σ).

S.V.Mach.

				Weak Mutation			**Strong Mutation**		
	δ	ϵ	**N**	Recall	$\overline{\textbf{time}}$	σ	Recall	$\overline{\textbf{time}}$	σ
ALE				100%	<0.01	<0.01	100%	<0.01	<0.01
BS	1e-10	0.01	1,897,519	98%	0.02	0.03	91%	0.26	1.00
	1e-10	0.10	18,975	97%	0.02	0.02	91%	0.04	0.06
	1e-05	0.10	9,764	97%	<0.01	0.02	91%	0.03	0.05
	0.10	0.10	2,396	98%	0.01	0.02	91%	0.02	0.04
RS	1e-10	0.01	1,897,519	97%	0.02	0.03	N/A	N/A	N/A
	1e-10	0.10	18,975	96%	0.01	0.02	N/A	N/A	N/A
	1e-05	0.10	9,764	97%	<0.01	0.01	N/A	N/A	N/A
	0.10	0.10	2,396	97%	0.01	0.03	N/A	N/A	N/A

C.P.Term.

				Weak Mutation			**Strong Mutation**		
	δ	ϵ	**N**	Recall	$\overline{\textbf{time}}$	σ	Recall	$\overline{\textbf{time}}$	σ
ALE				100%	<0.01	<0.01	100%	<0.01	<0.01
BS	1e-10	0.01	1,897,519	97%	0.49	9.05	91%	0.21	0.76
	1e-10	0.10	18,975	96%	0.02	0.10	91%	0.04	0.05
	1e-05	0.10	9,764	97%	0.01	0.05	91%	0.03	0.05
	0.10	0.10	2,396	96%	0.01	0.03	91%	0.03	0.04
RS	1e-10	0.01	1,897,519	97%	0.49	9.04	N/A	N/A	N/A
	1e-10	0.10	18,975	96%	0.02	0.11	N/A	N/A	N/A
	1e-05	0.10	9,764	97%	<0.01	0.05	N/A	N/A	N/A
	0.10	0.10	2,396	96%	0.01	0.04	N/A	N/A	N/A

Minepump

				Weak Mutation			**Strong Mutation**		
	δ	ϵ	**N**	Recall	$\overline{\textbf{time}}$	σ	Recall	$\overline{\textbf{time}}$	σ
ALE				100%	<0.01	<0.01	100%	<0.01	<0.01
BS	1e-10	0.01	1,897,519	98%	0.40	8.54	92%	0.21	0.80
	1e-10	0.10	18,975	98%	0.02	0.15	92%	0.04	0.06
	1e-05	0.10	9,764	99%	<0.01	0.04	92%	0.03	0.05
	0.10	0.10	2,396	98%	0.01	0.04	92%	0.03	0.04
RS	1e-10	0.01	1,897,519	98%	0.39	8.43	N/A	N/A	N/A
	1e-10	0.10	18,975	98%	0.02	0.15	N/A	N/A	N/A
	1e-05	0.10	9,764	98%	<0.01	0.06	N/A	N/A	N/A
	0.10	0.10	2,396	98%	0.01	0.05	N/A	N/A	N/A

Claroline

				Weak Mutation			Strong Mutation		
	δ	ϵ	**N**	Recall	**time**	σ	Recall	**time**	σ
ALE				100%	0.02	0.02	100%	0.10	0.12
BS	1e-10	0.01	1,897,519	99%	3.62	49.96	98%	0.59	2.00
	1e-10	0.10	18,975	99%	0.09	0.57	98%	0.17	0.42
	1e-05	0.10	9,764	99%	0.07	0.32	98%	0.17	0.28
	0.10	0.10	2,396	99%	0.05	0.12	98%	0.18	0.71
RS	1e-10	0.01	1,897,519	96%	29.99	139.34	N/A	N/A	N/A
	1e-10	0.10	18,975	95%	0.39	1.52	N/A	N/A	N/A
	1e-05	0.10	9,764	94%	0.23	0.80	N/A	N/A	N/A
	0.10	0.10	2,396	94%	0.10	0.25	N/A	N/A	N/A

Elsa-RR

				Weak Mutation			Strong Mutation		
	δ	ϵ	**N**	Recall	**time**	σ	Recall	**time**	σ
ALE				100%	<0.01	<0.01	100%	1.05	0.67
BS	1e-10	0.01	1,897,519	99%	0.06	0.05	95%	0.96	3.86
	1e-10	0.10	18,975	100%	0.04	0.04	95%	0.15	0.27
	1e-05	0.10	9,764	99%	0.05	0.04	95%	0.13	0.19
	0.10	0.10	2,396	100%	0.02	0.03	95%	0.09	0.16
RS	1e-10	0.01	1,897,519	88%	73.03	209.50	N/A	N/A	N/A
	1e-10	0.10	18,975	86%	0.92	2.56	N/A	N/A	N/A
	1e-05	0.10	9,764	86%	0.51	1.38	N/A	N/A	N/A
	0.10	0.10	2,396	87%	0.13	0.33	N/A	N/A	N/A

Elsa-RRN

				Weak Mutation			Strong Mutation		
	δ	ϵ	**N**	Recall	**time**	σ	Recall	**time**	σ
ALE				100%	0.01	0.01	100%	3.64	2.29
BS	1e-10	0.01	1,897,519	100%	0.05	0.05	90%	2.93	10.34
	1e-10	0.10	18,975	100%	0.04	0.04	90%	0.18	0.25
	1e-05	0.10	9,764	99%	0.04	0.04	90%	0.16	0.21
	0.10	0.10	2,396	100%	0.03	0.03	90%	0.10	0.11
RS	1e-10	0.01	1,897,519	97%	19.24	100.73	N/A	N/A	N/A
	1e-10	0.10	18,975	95%	0.37	1.42	N/A	N/A	N/A
	1e-05	0.10	9,764	95%	0.22	0.75	N/A	N/A	N/A
	0.10	0.10	2,396	94%	0.08	0.21	N/A	N/A	N/A

AGE-RR

				Weak Mutation			Strong Mutation		
	δ	ϵ	**N**	Recall	**time**	σ	Recall	**time**	σ
ALE				100%	0.64	0.94	100%	21.18	13.70
BS	1e-10	0.01	1,897,519	100%	0.06	0.08	90%	9.38	42.87
	1e-10	0.10	18,975	100%	0.05	0.10	90%	0.24	0.45
	1e-05	0.10	9,764	100%	0.04	0.08	90%	0.18	0.47
	0.10	0.10	2,396	100%	0.03	0.04	89%	0.09	0.25
RS	1e-10	0.01	1,897,519	96%	38.68	188.18	N/A	N/A	N/A
	1e-10	0.10	18,975	94%	0.68	2.50	N/A	N/A	N/A
	1e-05	0.10	9,764	95%	0.35	1.27	N/A	N/A	N/A
	0.10	0.10	2,396	94%	0.14	0.47	N/A	N/A	N/A

AGE-RRN

				Weak Mutation			Strong Mutation		
	δ	ϵ	**N**	Recall	**time**	σ	Recall	**time**	σ
ALE				100%	0.21	0.22	100%	75.29	51.92
BS	1e-10	0.01	1,897,519	100%	0.04	0.07	95%	7.10	32.32
	1e-10	0.10	18,975	100%	0.05	0.05	95%	0.32	0.46
	1e-05	0.10	9,764	100%	0.04	0.04	95%	0.27	0.43
	0.10	0.10	2,396	100%	0.04	0.04	95%	0.21	0.31
RS	1e-10	0.01	1,897,519	90%	117.21	362.41	N/A	N/A	N/A
	1e-10	0.10	18,975	88%	1.98	4.78	N/A	N/A	N/A
	1e-05	0.10	9,764	87%	1.12	2.63	N/A	N/A	N/A
	0.10	0.10	2,396	85%	0.41	0.87	N/A	N/A	N/A

Random 1

				Weak Mutation			Strong Mutation		
	δ	ϵ	**N**	Recall	**time**	σ	Recall	**time**	σ
ALE				100%	<0.01	<0.01	100%	448.61	339.19
BS	1e-10	0.01	1,897,519	100%	0.08	0.07	92%	0.78	2.50
	1e-10	0.10	18,975	100%	0.07	0.07	92%	0.09	0.08
	1e-05	0.10	9,764	100%	0.07	0.07	92%	0.09	0.06
	0.10	0.10	2,396	99%	0.07	0.07	92%	0.07	0.05
RS	1e-10	0.01	1,897,519	100%	0.03	0.07	N/A	N/A	N/A
	1e-10	0.10	18,975	100%	0.03	0.11	N/A	N/A	N/A
	1e-05	0.10	9,764	100%	0.03	0.09	N/A	N/A	N/A
	0.10	0.10	2,396	99%	0.03	0.08	N/A	N/A	N/A

Random 2

				Weak Mutation			Strong Mutation		
	δ	ϵ	**N**	Recall	**time**	σ	Recall	**time**	σ
ALE				100%	<0.01	<0.01	100%	412.37	168.90
BS	1e-10	0.01	1,897,519	100%	0.11	0.06	89%	1.22	3.35
	1e-10	0.10	19,975	100%	0.10	0.06	89%	0.14	0.09
	1e-05	0.10	9,764	100%	0.11	0.07	89%	0.14	0.08
	0.10	0.10	2,396	100%	0.11	0.06	89%	0.12	0.07
RS	1e-10	0.01	1,897,519	100%	0.04	0.10	N/A	N/A	N/A
	1e-10	0.10	18,975	100%	0.03	0.07	N/A	N/A	N/A
	1e-05	0.10	9,764	100%	0.03	0.07	N/A	N/A	N/A
	0.10	0.10	2,39	99%	0.03	0.09	N/A	N/A	N/A

Random 3

				Weak Mutation			Strong Mutation		
	δ	ϵ	**N**	Recall	**time**	σ	Recall	**time**	σ
ALE				100%	<0.01	<0.01	100%	367.99	154.80
BS	1e-10	0.01	1,897,519	100%	0.11	0.06	91%	1.04	3.20
	1e-10	0.10	18,975	100%	0.09	0.04	91%	0.23	0.15
	1e-05	0.10	9,764	100%	0.09	0.04	91%	0.23	0.14
	0.10	0.10	2,396	100%	0.09	0.05	91%	0.19	0.12
RS	1e-10	0.01	1,897,519	100%	0.03	0.10	N/A	N/A	N/A
	1e-10	0.10	18,975	100%	0.03	0.16	N/A	N/A	N/A
	1e-05	0.10	9,764	99%	0.03	0.12	N/A	N/A	N/A
	0.10	0.10	2,396	99%	0.02	0.07	N/A	N/A	N/A

Random 4

				Weak Mutation			Strong Mutation		
	δ	ϵ	**N**	Recall	**time**	σ	Recall	**time**	σ
ALE				100%	<0.01	<0.01	100%	306.37	127.02
BS	1e-10	0.01	1,897,519	100%	0.11	0.06	91%	1.09	3.23
	1e-10	0.10	18,975	100%	0.10	0.05	91%	0.22	0.14
	1e-05	0.10	9,764	100%	0.10	0.05	91%	0.23	0.12
	0.10	0.10	2,396	100%	0.09	0.04	91%	0.19	0.11
RS	1e-10	0.01	1,897,519	100%	0.04	0.25	N/A	N/A	N/A
	1e-10	0.10	18,975	99%	0.03	0.25	N/A	N/A	N/A
	1e-05	0.10	9,764	100%	0.03	0.10	N/A	N/A	N/A
	0.10	0.10	2,396	99%	0.02	0.09	N/A	N/A	N/A